Comparing Three Approaches to Transformational Programming[*]

— Preliminary Version —

Konstantin Läufer
Courant Institute of Mathematical Sciences
New York University
715 Broadway, 7th floor
New York, New York 10003, USA
laufer@cs.nyu.edu

April 22, 1991

Abstract

Transformational programming is a methodology that intends to formalize the development of programs from problem specifications. Given the recent effort towards the design of a common prototyping system (CPS) for the Ada programming language, transformation systems may be reconsidered as possible components of prototyping systems. This paper examines and evaluates three approaches to transformational programming:

- The Munich CIP project (Computer-aided, Intuition-guided Programming) consists of a strongly typed, wide-spectrum language with user-defined algebraic types and a semi-automatic transformation system that requires user guidance.
- By contrast, "Algorithmics," the work on algebraic specification originating from IFIP WG 2.1, is a pure pencil-and-paper approach to transformational programming. It provides a concise, uniform mathematical notation and includes work on nondeterminism.
- RAPTS (Robert A. Paige's Transformation System) is a fully mechanical system that transforms high-level specifications to C code. The specifications are given in a functional subset of SETL augmented with fixed-point operations.

First, we describe each system in detail and highlight interesting features. Next, we establish a framework of common criteria by which such different transformational systems can be evaluated. Finally, we point out common features and differences of the three systems and compare them.

[*] Supported in part by the Defense Advanced Research Projects Agency/Information Systems Technology Office under the Office of Naval Research contract N00014-90-J-1110

1 Introduction

Programming is a complex task; large programs are difficult to produce, and once produced they are often unreliable and expensive to maintain or improve. The "software crisis" remains an important issue in software engineering. A large amount of research has been going on to overcome this crisis: On one hand, a considerable portion of the work has focused on *formalizing* and *automating* the software development process; on the other hand, *prototyping* has been promoted as a method of obtaining information about a problem before implementing it in a production language.

Traditionally, programs have been verified *experimentally* by choosing a number of input test cases, running the program, and evaluating the results. The problem with this approach is that it can never formally prove the correctness of the program. There might always be cases not captured by the test data. *Prototyping* attempts to catch errors at an early a stage of the program development.

Formal verification, on the other hand, is an *analytic* process that defines in mathematical terms what it means for a program to be correct, given a mathematical definition of the programming language used, and a formal specification of the problem being solved. However, the formal verification of large programs has turned out to be impractical.

By contrast, the transformational approach (see [PS83] for an excellent survey) is a *synthetic* or *constructive* one. A program is derived from a problem specification by successive application of correctness-preserving transformations that lead to a correct implementation of the problem. A key point is the *reusability* of transformation rules; once a rule is proven correct, it can be used again if applicable to the particular situation. Libraries of known transformations can be established; as the libraries grow, the number of new proofs required in the development of a program decreases.

An advantage of transformational programming is the tight coupling of program development and verification. A complete program consists of the specification, the final executable program, and the complete history of intermediate transformations. This history serves as a program documentation, since it captures all ideas and decisions relevant for the program. Later modification is facilitated because it corresponds to stepping back in the development and following an other branch at a point where a decision was made.

The transformational paradigm leaves a lot of room for automation; the programmer can be relieved of tedious tasks that come up during program development such as rewriting program pieces, verifying applicability conditions of transformations, reviewing the development history etc. However, important design decisions are generally left to the programmer.

Recently, a considerable effort has been made towards the development of a common prototyping system (CPS, see [Gab88]) for the Ada language. As a future goal, we would like to explore how a transformational component can be incorporated into such a prototyping environment. The transformational system could be used to aid replacing pieces of the prototype by more efficient pieces obtained by transforming into a target language.

From this perspective, we take a detailed look at several approaches to transformational programming in the next three sections, emphasizing relevant features. In the last section, we present a framework of common criteria for the comparison of transformational systems. Finally, we apply these criteria to the systems under consideration.

2 The Munich CIP Approach

The CIP project was created with the goal of producing an implemented software development system using the transformational paradigm. CIP stands for "Computer-aided, Intuition-guided

Programming". The project consists of two components: The wide-spectrum programming language CIP-L [B$^+$85a], and the program-transformation system CIP-S [B$^+$87a]. The user of CIP-S *specifies* a problem in CIP-L and *derives* a (hopefully efficient) *implementation* of the problem.

A *wide-spectrum language* is a single formal language which includes specification constructs, implementation constructs, and any intermediate constructs needed for program development. The use of such a language makes it easy to formulate program transformations as local, correctness-preserving, source-to-source transformations on a common semantic basis. CIP-L is a *scheme language*, i. e., a language without a fixed set of basic data types. This makes it possible to manipulate program schemes which can later be instantiated as needed.

The system CIP-S is an interactive system whose purpose is to assist the programmer in clerical work and other tasks that can be automated. It performs routine task such as keeping track of various versions of a program, and it provides a mechanism for performing transformations. In addition, it helps the user verify the applicability of a transformation to a portion of the program. It also maintains a library of transformation rules and assists in searching the library. Decisions, however, are always made by the user; he or she supplies the intuition.

It is interesting to note that the CIP system was developed using exactly the CIP methodology. First, a formal specification of CIP-S was written, which includes a logical calculus for program transformations. Then, a prototype of the system was derived from the specification using program transformations. The system has not yet reached its final state; therefore parts of its description below refer to the specification rather than the implemented system.

For a detailled evaluation of the CIP project see [WJS87].

2.1 The System CIP-S

The purpose of CIP-S is the transformational development of program schemes. This includes manipulation of actual programs, derivation of new transformation rules within the system, transformation of algebraic types and type schemes, and verification of applicability conditions.

Transformation rules in CIP consist of three components: input template, output template, and applicability condition. The input and output templates are *program schemes*, possibly including *context parameters*, which allow the user to mark fragmentary terms (*contexts*) as replaceable. A typical rule, for example, expresses the distributivity of the conditional expression over varying contexts:

$$\begin{array}{c} F[\textbf{if } C \textbf{ then } A \textbf{ else } B \textbf{ fi}] \\ \updownarrow \\ \textbf{if } C \textbf{ then } F[A] \textbf{ else } F[B] \textbf{ fi} \end{array}$$

Under this rule, the code fragment

$$x \ := \ \textbf{if } x \geq 3 \textbf{ then } 4 \textbf{ else } 2 \textbf{ fi}$$

is transformed to

$$\textbf{if } x \geq 3 \textbf{ then } x \ := \ 4 \textbf{ else } x \ := \ 2 \textbf{ fi}.$$

In addition to such rules, CIP provides a metalanguage for *transformation expressions*, which allows transformational algorithms to be expressed similarly to regular expressions.

The transformation rules are organized as *generative sets*, i. e., a small set of powerful elementary transformations that can be used to construct new rules. The basic rules include general principles (such as unfold, fold), the definitional rules of the language, and the axioms and inference rules of the predefined types. Furthermore, some frequently used derived rules, as well as standard implementation techniques for some abstract types, are available. CIP-S allows the user to define new transformation rules and maintain libraries of rules.

CIP-S records the whole development of a program from a specification. It allows the user to browse through the development, backtrack and continue with another program version, return to previous development steps etc. The system assists the user in the selection and application of transformation rules. All this calls for a sophisticated interactive user environment on top of the system core. This environment makes use of facilities such as graphic screens, windows, and mice, and allows the user to perform all the system functions conveniently.

There are several languages involved in the design of CIP-S. As the *object language*, the language for formulating program schemes, an algebraically defined language is assumed. CIP-L was designed especially to serve this purpose. Transformation algorithms are formulated in a special metalanguage, which consists of constructs similar to regular expressions, over transformation rules as described above. This language is supposed to be later replaced by a simple applicative language. Finally, the language for enabling conditions of transformation rules consists of predicates over terms of the object language.

Note that CIP-S is *extensible* with respect to the transformation libraries, object language and user environment. Since CIP-S is being used for its own development, it is even possible to change the implementation language by going back to the stage before fixing the language, and changing to a different language.

2.2 The Language CIP-L

In order to accommodate a full variety of programming styles, the CIP group conceived the notion of a wide-spectrum language, a single language providing all the styles required, from specification level down to implementation level. CIP-L has the following components:

1. Algebraic or abstract data types. Types provide the means for specifying basic objects and operations on those objects. They determine the application for the language.

2. The scheme language. This language is the heart of CIP-L and provides the means for writing specifications and algorithms. It is structured hierarchically with levels from specification through application, procedure down to control. The specification level is called *kernel* and consists of expressions over the basic types.

3. Programs. Programs connect the first two parts of the language. A program is a finite set of components, which may be types, computation structures, modules or devices. Computation structures, modules and devices are implementations of the basic data types underlying the scheme language.

The language is defined formally. Algebraic data types are defined through algebraic semantics. Only the kernel of the scheme language is defined through traditional denotational semantics [Sto77,Gor79]. The additional levels are defined in terms of the previous levels by *transformational semantics*.

CIP-L has an abstract syntax, hence its external representation is flexible. The language can be represented by an ALGOL-like or Pascal-like variant, or even by a LISP-like or Prolog-like variant, depending on what the particular application calls for. All that is required is an appropriate parser/unparser.

Other notable characteristics of CIP-L are *full typing*, *modularity*, provided by algebraic types on the specification level and by computation structures, modules and devices (modules with internal state) on the implementation level, and *nondeterminism.*

It is not surprising that some modern prototyping languages are designed similarly; for example, the prototyping language Griffin [$\mathrm{D}^{+}90$], currently under development at New York University, shares several key features with CIP-L: strong typing, user-defined algebraic data types, and a layered semantics.

We will now examine the three components of CIP-L in detail.

2.3 Abstract Data Types

In programs certain identifiers are used for object sets (sorts), for elements of these sets (constants), and for functions operating on them (operations). Before we give an interpretation to these symbols, we are dealing with program schemes. The purpose of abstract types is to give an interpretation by presenting the sorts, constants and operations with their functionality and by stating the properties in form of axioms (laws). Abstract types may be constructed hierarchically.

An abstract type, briefly called type, consists of two main parts:

- The *signature*, which is a list of symbols whose meaning is specified within the type. Each symbol is associated with a specification of its kind (sort or carrier set of an abstract type): Symbols for sorts are given as attributes the keyword **sort**, symbols for constant elements their sort, and operation symbols are given their functionality. A subset of these symbols is made visible to the outside by the *list of constituents* of the type. Symbols not made visible are called *hidden* symbols. The signature determines a language of well-formed *terms* (formed from the constants by applying the operations) which may also contain free identifiers.

- The collection of *laws*, which specify the properties of the symbols. The laws are first-order logic formulas built from equations and inequalities and logical operators $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, and of the quantifiers $\forall$, $\exists$, over sorts of the type.

The meaning of a type T is defined to be the class of all term-generated models of T. Note that there might be many different models for a type T, or there might be only one, depending on the particular set of laws supplied.

For example, the following type specification describes integer arithmetic modulo 3:

```
type MOD3 ≡
        mod3, zero, one, two, succ :
        sort mod3,
        mod3 zero, one, two,
        funct (mod3) mod3 succ
laws mod3 x, y :
        zero ≠ one,
        zero ≠ two,
        one ≠ two,
        succ(zero) = one,
        succ(one) = two,
        succ(two) = zero
end of type
```

It has the visible constituents **mod3**, *zero*, *one*, *two* and *succ*, where *zero*, *one* and *two* are constants of sort **mod3**, and *succ* is a function taking one **mod3** parameter and returning a **mod3**.

A type T may depend on other types in three ways:

type inclusion: A type T can be made available by an instantiation, possibly with renaming, using an **include** clause. Such an instantiation is equivalent to the textual substitution of the body of T (with consistent renaming); there is no protection on the models of T. Instantiations are simply a shorthand notation for type bodies; they have no independent semantics.

base types: A type may use another, "primitive" type by means of a **based on** clause. In this case, the primitive type is protected against modifications of the carrier sets of its models. This is called *hierarchy preservation* and guarantees independent implementability of the primitive type. The types mentioned in the **based on** clause should be thought of as part of the new type.

type parameters: Type specifications may be parametrized. The parameters can be sort symbols, constant symbols, and operation symbols. They are attributed in the same way as within a signature. Such type specifications are called *type schemes* or *generic types*. They may be instantiated both in **based on** clauses and **include** clauses.

Modes are a shorthand notation for certain types and type schemes which are frequently used. Modes are syntactically introduced by (possibly recursive) mode declarations. The semantics of mode declarations is explained by instantiation of the associated types; thus recursive modes are explained in a straightforward manner. There are two basic kinds of mode declarations:

sum: A sum is a disjoint union of a finite number of variants and comes with constructors, projections, and boolean test functions, which indicate whether an intended projection would yield a defined result.

product: A product is a finite, heterogeneous tuple. A product has a constructor function and component selector functions.

For practical purposes, the programming system for the language should provide a collection of *predefined types* as a basis for further programming activities. Typical standard types include boolean values, integers, finite sets and multisets, finite mappings, sequences, trees, pointers and pointer structures.

2.4 The Scheme Language of CIP-L

We will present the different style levels of the scheme language and give examples. For most examples, we will use an ALGOL-like representation.

2.4.1 The Expression Language for Logic and Functional Programming

An expression denotes objects of a certain kind (e. g., a sort of an algebraic type). Fundamental expressions are the terms over basic object and operation symbols of an underlying algebraic type; these symbols have to be interpreted in some model of that type. Other examples of expressions follow:

Guarded Expressions. The guarded expression has the following form:

$$\textbf{if } B_1 \textbf{ then } E_1 \; [\!] \; \ldots [\!] \; B_n \textbf{ then } E_n \textbf{ fi}$$

The *guards* B_i are boolean expressions, and the E_i are expressions of the same non-functional kind $\mathbf{m}$. The guarded expression is nondeterministic, its value is one of the E_i for which the corresponding B_i is true. The set of possible values is called *breadth.* If none of the B_i is true, the value of the guarded expressions is not defined. If the breadth contains only one value, the expression is called *determinate*, otherwise it is called *nondeterminate.*

If B is determinate, the guarded expression

$$\textbf{if } B \textbf{ then } E_1 \; [\!] \; \neg \; B \textbf{ then } E_2 \textbf{ fi}$$

is semantically identical to the *conditional expression*

$$\textbf{if } B \textbf{ then } E_1 \textbf{ else } E_2 \textbf{ fi}$$

A guarded expression with constantly true guards describes an arbitrary choice between the branches. Its breadth is the union of the breadth of E_1 and the breadth of E_2. Such a guarded expression may be abbreviated by

$$(E_1 \; [\!] \; E_2),$$

an expression called *finite choice.* The finite choice has the following property with respect to the application of a function f: The expressions $f((E_1 \; [\!] \; E_2))$ and $(f(E_1) \; [\!] \; f(E_2))$ have the same breadth. This holds since we have call-by-value and call-time choice semantics. To avoid semantic problems, we do not allow the nondeterministic choice between higher-order functions.

Function Abstraction and Application. An *abstraction* is a parameterized expression; it is of the form

$$(\mathbf{m}_1 \; x_1, \ldots, \mathbf{m}_n \; x_n) \; \mathbf{r} : \; E$$

where the x_i are *parameters* of kinds $\mathbf{m}_i$ and may occur free in the expression E. If f is such an abstraction, then the *application* of f to expressions E_i of appropriate kinds is expressed by

$$f(E_1, \ldots, E_n).$$

The functionality (kind) of f is given by

$$\textbf{funct } (\mathbf{m}_1, \ldots, \mathbf{m}_n) \; \mathbf{r}.$$

We may restrict the domain of arguments by putting an appropriate *assertion* behind the formal parameter list, as in

$$(\textbf{nat } a, \textbf{nat } b : \; a \; \geq \; b) \; \textbf{nat} : \; a \; - \; b.$$

If the assertion is not satisfied for the actual parameters, the result of the application is undefined.

Note that the language is fully functional; functions may occur as parameters and as results of functions. There are two standard higher-order operations: function composition $\circ$ and function tupling (as in FP).

Fixpoints and Recursion. A function may be defined as a fixpoint of a functional equation. We have the fixpoint operator $\mathbf{Y}$ which is applied to a pair consisting of a function identifier and an abstraction with a free occurrence of that identifier. The application of the fixpoint operator gives the minimal solution of the corresponding functional equation. Consider, for example, the following definition of the factorial:

```
(Y f   : (nat n) nat :
         if n = 0 then 1
         else n * f(n - 1)
         fi
)
```

Here, f is not known outside of the fixpoint expression and cannot be used as a function identifier.

Note that $(\mathbf{Y}\ f\ :\ A)$ reduces to A itself if there is no free occurrence of f in A. Furthermore, the fixpoint operator generalizes to systems of functional equations.

Descriptive Constructs. If P is a *characteristic* predicate, then the object specification

$$\mathbf{that}\ \mathbf{r}\ x :\ P(x)$$

denotes the unique x of kind $\mathbf{r}$ for which $P(x)$ holds. This form is called *description.*

If P is not a characteristic predicate, we may still write

$$\mathbf{some}\ \mathbf{r}\ x :\ P(x).$$

Here the breadth of this *comprehensive choice* depends on P.

We may form sets by *set comprehension*:

$$\{\mathbf{m}\ x :\ P(x)\}$$

denotes the set of all objects of kind $\mathbf{m}$ that satisfy $P(x)$. (We may also enumerate sets; enumeration can be viewed as a comprehension).

Expressions may be quantified using the *universal quantifier* $\forall$, the *existential quantifier* $\exists$ and the *unique existential quantifier* $\exists_1$. Quantified expressions have boolean results and are typically used in conditional expressions.

2.4.2 The Full Applicative Language

This level of the language provides facilities for the declaration of objects and functions.

Object Declarations. We may give objects names by means of a *collective object declaration*; we write

$$(\mathbf{m}_1\ x_1, \ldots, \mathbf{m}_n\ x_n)\ \equiv\ (E_1, \ldots, E_n)$$

and introduce the *object identifiers* x_i. For $n = 1$, the parentheses are omitted; we simply write

$$\mathbf{m}\ x\ \equiv\ E.$$

We may *restrict* object declarations in a similar way as function parameters; consider e. g.,

$$(\mathbf{nat}\ a, \mathbf{nat}\ b :\ a\ \geq\ b)\ \equiv\ (19, 7).$$

Naturally, declarations may occur sequentially.

Function Declarations. If $\mathbf{m}$ is a function kind in an object declaration, we have a function declaration in λ-calculus style (E is an abstraction in this case).

Functions may be declared in the ALGOL style by writing

$\textbf{funct}\ f\ \equiv\ (\mathbf{m}\ x)\ \mathbf{r}:\ E.$

This is just an abbreviation for convenience. Similarly, a unary fixpoint operator may be suppressed by replacing

$\textbf{funct}\ (\mathbf{m})\,\mathbf{r}\,f\ \equiv\ (\mathbf{Y}\,f:\ (\mathbf{m}\ x)\ \mathbf{r}:\ E)$

by the traditional ALGOL-style way of declaring a recursive function:

$\textbf{funct}\ f\ \equiv\ (\mathbf{m}\ x)\ \mathbf{r}:\ E,$

2.4.3 The Procedural Language

This level of CIP-L contains constructs such as variables, assignments and procedures.

Variables and Assignments. Similar to the object declaration, we may declare a *variable*, which may be initialized:

$\textbf{var}\ \mathbf{m}\ x\ :=\ E.$

We assign a value to a variable using the *assignment statement*

$x\ :=\ E$

with the semantics of variables as *reusable* object identifiers.

Variables can be declared and initialized collectively, as in

$(\textbf{var int}\ i, \textbf{var int}\ j)\ :=\ (3,4),$

and their values may be changed by a *collective assignment*, e. g.,

$(i,j)\ :=\ (7,13).$

Statements are separated using ";".

Procedures. Procedures are different from functions in that they do not yield a result. In CIP-L, expressions are strictly distinguished from statements to keep the collection of program transformations manageable. Hence there are only pure procedures. Procedures either explicitly change formal variable parameters or implicitly modify nonlocal variables (suppressed variable parameters). Every procedure call can be reduced to an assignment to the variable parameters.

Procedures are declared as in the following example:

$\textbf{proc}\ \mathit{assigntwo}\ \equiv\ (\textbf{var int}\ i):i\ :=\ 2,$

and are called in a *call statement*, as in

$\textbf{var int}\ k;\ \textbf{call}\ \mathit{assigntwo}(k).$

To avoid aliasing, it is required that no two (explicit or implicit) variable parameters of a procedure may be associated with the same variable.

2.4.4 The Control-oriented Language

This part of CIP-L consists of the control-flow constructs such as the usual *iteration statements* (**while** loop, **do** loop with explicit **leave** statement in the loop body). It also contains *labels* and **goto** statements.

2.4.5 Parallel Constructs

CIP-L allows limited parallelism in the form of the parallel composition of blocks. Critical regions may be constructed using the following form:

await C **then** P **endwait**,

where C is a condition, and P is the critical region.

2.5 Programs in CIP-L

Computation structures, modules, and devices provide interpretations of the algebraic types that underly the scheme language. They can be grouped together as *components* of a *program.* The program is executed by invoking executable visible constituents of its components.

Computation Structures. A *computation structure* is a collection of declarations for sorts, objects, and functions that are made available to the outside. Computation structures provide a means for implementing types. The declaration of a computation structure has the form

```
structure CS ≡ ≪ constituents ≫
      D_1, ..., D_r
end of structure
```

The *list of constituents* corresponds to that of a type; it consists of symbols for sorts, constants, and functions visible to the outside. Since computation structures are intended as implementations for types, procedure identifiers must not appear in the list of constituents. Computation structures may be parameterized and thus used as "implementation schemes".

The *body* $D_1, \ldots, D_r$ of the structure provides definitions at least for the constituents; further "hidden" entities may be defined for internal use. The kinds of definitions that may appear in structures include:

- instantiations of types, type schemes, (parameterized) structures, and modules
- declarations of modes, functions, objects, and procedures without global variables

The following example illustrates how a type *NEWBOOL* might be implemented:

```
structure BOOLIMPL ≡ newbool, true, false, and, or, not :
      mode newbool ≡ false | true,
      funct and ≡ (newbool x, y) newbool :
            if x = true ∧ y = true then true else false fi,
      funct or ≡ (newbool x, y) newbool :
            if x = false ∧ y = false then false else true fi,
      funct not ≡ (newbool x) newbool :
            if x = true then false else true fi
end of structure
```

A computation structure is called a *syntactically correct* implementation of a type if the elements of the respective constituent lists together with their sorts and functionalities coincide (after consistent renaming). A syntactically correct implementation of a type is called *semantically correct* if the implementation provides a model of the type.

Modules and Devices. *Modules* and *devices* are similar to computation structures, except that they may also export procedures. Devices may, in addition, contain definitions of hidden variables which can be manipulated through the procedures provided. An example for a module would be a collection of procedures manipulating a stack given as a parameter. One could imagine a device which supplies similar functions, but has the stack as an internal variable accessible only through the procedures.

3 Algorithmics — an Algebraic Approach

The Algorithmics group (IFIP WG 2.1) is motivated by the conviction that a great deal of the activities of algorithmic development should and can be performed in a similar way as mathematical activities. The group is developing a language for "algorithmic expressions" with the idea that algorithms are developed by manipulating such expressions [Mee83,Mee84,B$^+$85b,B$^+$87b]. The key point is that the language should be a uniform framework rather than a union of a specification language and an implementation language: it must be possible to view all expressions as specifications, although not all expressions need to suggest an implementation. The language should be comparable to the language used by mathematicians, with notations that give a convenient way to express concepts and facilitate reasoning. Clearly, a nice algebraic structure is a prerequisite for obtaining interesting results, since otherwise, no general laws can be expressed, and each step has to be verified afresh.

It is important to note that the specific notational conventions used should be given little weight. The idea is to find the right balance between readability, terseness, and dependability. In our presentation of the framework, we try to use a "conventional", self-explanatory notation using parentheses only to avoid ambiguities.

3.1 Structures

First of all, we need to define some objects to work on. Suppose D is a domain, e. g., numbers or booleans. We define a new domain

$$S_D \; = D \oplus S_D \; \times \; S_D,$$

the domain of D-structures, each of which is either an element of D or constructed from two D-structures.

To actually build D-structures, we use an *injection operation* ˆ and a *construction operation* $+$. If a is an element of D, then $\hat{}a$ stands for the corresponding element of S_D. We shall write $\hat{a}$ instead of $\hat{}a$. If x and y are D-structures, then $x + y$ denotes the D-structure constructed from x and y. S_D is the set of all structures that can be built from D by a finite number of injections and constructions.

We can introduce an *identity element* by redefining

$$S_D \; = D \oplus \{0\} \oplus S_D \; \times \; S_D$$

and imposing the *identity law*

$$x + 0 \; = \; 0 + x \; = \; x.$$

Now we have a reasonable starting point, since we can obtain familiar structures by imposing other algebraic laws. Of particular interest are the laws of *associativity*:

$$x + (y + z) = (x + y) + z,$$

of *commutativity*:

$$x + y = y + x,$$

and of *idempotency*:

$$x + x = x.$$

It is interesting to note that with each new law we get another familiar data structure: we get, successively, *lists, multi-sets and sets.* For sets, $\hat{}$ is the function $a \mapsto \{a\}$, $+$ is the set union $\cup$, and 0 is the empty set.

3.2 Basic Operations

We will use the following lemma to develop further results:

Lemma 3.1 (Induction Lemma) *Let f and g be two functions defined on S_D, satisfying the following conditions:*

(i) $f\ 0 = g\ 0$

(ii) $f\ \hat{a} = g\ \hat{a}$*, and*

(iii) $f\ x = g\ x$ *and* $f\ y = g\ y$ *as induction hypothesis implies* $f(x + y) = g(x + y)$

Then $f = g$.

Proof. By induction on the complexity of the function argument.

Note that the first part of the lemma can be omitted if S_D does not have an identity.

Let us introduce the following basic operations:

Map. The operator $*$ applies a function to each "member" (elementary component) of its argument, and the result is a structure of the function values obtained. If f is a function, then $f\ *$ stands for the function satisfying

(i) $f * 0 = 0$

(ii) $f * \hat{a}\ =\ \hat{}\, f\ a,$

(iii) $f * (x + y)\ = f * x + f * y.$

Filter. The operator $\triangleleft$ takes a predicate p and a structure x and returns the structure of components of x that satisfy p. $p\ \triangleleft$ stands for the function satisfying

(i) $p \triangleleft 0 = 0$

(ii) $p \triangleleft \hat{a} = \begin{cases} \hat{a} & \text{if } p\ a \\ 0 & \text{otherwise} \end{cases}$

(iii) $p \triangleleft (x + y) = (p \triangleleft x) + (p \triangleleft y)$

Reduce. If $\oplus$ is a binary operation in D, and x is in S_D, then $\oplus/\ x$ returns the value obtained by inserting $\oplus$ between adjacent components of x. $\oplus/$ is the function satisfying

(i) if $\oplus$ has an identity element e such that $e \oplus a = a \oplus e = a$, then $\oplus/0 = e$,

(ii) $\oplus/\ \hat{a} = a$, and

(iii) $\oplus/\ (x + y) = (\oplus/\ x) \oplus (\oplus/\ y)$.

Clearly, $\oplus$ must be associative for $\oplus/$ to be unique.

Now that we have some basic operations, we can formulate a few laws. First, $*$ distributes through $+$; for all structures x and y we have

$$f * (x + y) = (f * x) + (f * y).$$

Second, $*$ distributes backwards through functional composition:

$$(f \circ g) * \ = (f\ *) \circ (g\ *).$$

Furthermore, if f is injective with inverse f^{-1}, then

$$(f\ *)^{-1} = (f^{-1}\ *).$$

We have some rules involving $\triangleleft$. Filtering is commutative:

$$p \triangleleft q \triangleleft x = q \triangleleft p \triangleleft x.$$

$p\ \triangleleft$ is an idempotent operation, i. e.,

$$p \triangleleft p \triangleleft x = p \triangleleft x.$$

Finally, the following commutativity relation holds between $*$ and $\triangleleft$:

$$p \triangleleft f * x = f * (p \circ f) \triangleleft x.$$

The proofs of these laws are straightforward applications of the induction lemma.

3.3 Homomorphisms

A homomorphism is essentially a linear operation with respect to $+$. A function h is a homomorphism in $S_{D_1} \rightarrow S_{D_2}$ if there exists an associative operator $\oplus$ with identity element e such that

(i) $h\ 0 = e$ and

(ii) $h(x + y) = h\ x \oplus h\ y$

If $h\ 0$ is not defined, then $\oplus$ need not have an identity element.

Note that this gives an algebraic formulation of the "Divide and Conquer" paradigm; part (ii) tells us that to conquer a compound structure z, we can divide it in two parts x and y, conquer these parts, and combine the results.

It is worthwhile to examine homomorphisms, since they represent a general class of operations, of which $f\ *$ and $\oplus/$ are special cases. By combining them in the form $(\oplus/) \circ (f\ *)$, all such homomorphisms can be expressed. This can be stated in form of another lemma:

Lemma 3.2 (Homomorphism Lemma) *A function h is a homomorphism iff $h = (\oplus/) \circ (f*)$ for some operator $\oplus$ and function f.*

Proof. The "only if" part can be shown using the distributive laws for $*$ and $/$, and the "if" part is an application of the induction lemma.

We can derive several useful identities as a consequence of the Homomorphism Lemma. They generalize the distributive laws of $*$, $\triangleleft$ and $/$.

Lemma 3.3 (Domain Switching) *Let function f, operations $\oplus$ and $\otimes$ satisfy $f(a \oplus b) = (f\ a) \otimes (f\ b)$ and $f\ \oplus/0 = \otimes/0$.*
Then $f\ \circ (\oplus/)\ = (\otimes/)\ \circ(f\ *)$.

Proof. Let $g = f \circ (\oplus/)$ and apply the homomorphism lemma.

Lemma 3.4 (Promotion) *For arbitrary function f, predicate p and associative operator $\oplus$ we have:*

($*$-promotion) $(f\ *) \circ (+/) = (+/) \circ ((f\ *)*)$

($\triangleleft$-promotion) $(p\ \triangleleft) \circ (+/) = (+/) \circ ((p\ \triangleleft)*)$

($/$-promotion) $(\oplus/) \circ (+/) = (\oplus/) \circ ((\oplus/)*)$.

Proof. We first prove $*$-promotion and $/$-promotion by the homomorphism lemma. We then use these results to show $\circ$-promotion.

Note that each of these laws corresponds to a whole set of program transformations. The promotion laws say that rather than mapping, reducing or filtering over one large structure, one can divide the structure into smaller ones, map reduce or filter each of these, and combine the results.

3.4 Selection

An important class of operations are *selection operations*, which select one out of two values. Minimum and maximum are such operations:

Minimum, Maximum. $a \downarrow b$ selects the smaller of a and b, and $a \uparrow b$ selects the larger.

If we now write $\downarrow/\ x$ for some structure x, we get the smallest component of x. A problem can arise if $x = 0$. 0 does not contain any components, hence $\downarrow/0$ is undefined. This can be fixed by introducing a *fictitious value* ∞. Such a domain extension can drastically simplify an algorithmic expression, since it reduces the number of special cases to be considered. However, it may introduce inconsistencies with additional laws or with laws involving other operations on the domain. To give an example of the possible inconsistencies, consider the operation $\ll$ defined by

$$a \ll b = a.$$

This selection operation is associative, since $(a \ll b) \ll c = a \ll (b \ll c) = a$. The function $\ll/$ selects the first element of a list (or the leftmost element of a tree). Now consider $\ll/0$, where 0 is the empty list. Then $(\ll/0) \ll a = a$, since $\ll/0$ is the identity element of $\ll$. But from the definition of $\ll$, we know that $(\ll/0) \ll a = \ll/0$. Hence $a = \ll/0$ for arbitrary a. The problem arises since the law $a \ll b = a$ already involves the identity element of $\ll$, in fact, each element is a right identity element of $\ll$. To resolve this inconsistency, we can either restrict the law $a \ll b = a$ to $a \neq \ll/0$, or use $\ll/0$ as a right identity only. Which solution is better depends on the particular application.

Many programming problems can be formulated as optimization problems: find the smallest, largest or cheapest in some given class of values. Such problems can be specified using the following operation:

Optimize. If f is a function, $a \downarrow_f b$ selects either a or b according to which is smaller, $f\ a$ or $f\ b$. The definition of $\uparrow_f$ is analogous, it selects a or b depending on which is greater, $f\ a$ or $f\ b$. We have

$$a \downarrow_f b = \begin{cases} a \text{ if } f\ a < f\ b \\ b \text{ if } f\ a > f\ b. \end{cases}$$

What happens in the case $f\ a = f\ b$? If f is an injective function, then $a = b$, and we return that value. However, in a lot of cases the function f is not injective. Then it makes no sense to say "let $a \downarrow_f b$ be *the* element in the set $\{a, b\}$ minimizing f", instead we should say "let $a \downarrow_f b$ be *some* element in the set $\{a, b\}$ minimizing f". Thus $\downarrow_f$ contains some nondeterminism. In developing solutions to optimization problems, is is desirable to allow nondeterminism, since we are not normally interested in any other property of the result than that it minimizes f.

3.5 Nondeterminism

In the following, we will examine nondeterminism in more detail. As a matter of fact, the members of the Algorithmics group have different views as to what exact approach to nondeterminism should be taken. We discuss three possible approaches [Mee83,B$^+$85b].

One possibility is avoiding explicit nondeterminism and allowing under-specification only through set construction. In this approach, one formulates the set of all solutions to a problem, and then selects a particular member of this set by some further step. The main advantage of this method is that the semantics of the expression language is simpler. On the other hand, the objects and functions in consideration become more complicated; instead of equality predicates it becomes necessary to go to set membership.

As a second possibility, we can allow a choice operator $[\!]$ into the notation, but let it always denote some definite, but unspecified, operator. The only property of $[\!]$ one may assume is that $[\!]$ is selective. More precisely,

$$a \mathbin{[\!]} b = sel(a, b)$$

for some selection function sel which is to be specified later. This model is semantically simple, but note that all different occurrences of $[\!]$ must be bound to the same sel. Hence certain, laws which seem obvious may contradict each other. For instance, the laws

$$a \mathbin{[\!]} b = b \mathbin{[\!]} a$$

cannot be valid together with

$$f(a \mathbin{[\!]} b) = (f\ a) \mathbin{[\!]} (f\ b).$$

To see this, chose f, a and b such that $a \neq b, f\ a = b$ and $f\ b = a$.

The third approach is to have an operator $[\!]$ denoting arbitrary choice. The following laws involving $[\!]$ are desirable, but mutually incompatible:

(i) A well-defined notion of a reflexive, transitive refinement operation ($\Rightarrow$) such that all constructs are monotonic with respect to $\Rightarrow$.

(ii) The property that $x \Rightarrow y$ iff $x \mathbin{[\!]} y = x$.

(iii) The laws that $[\!]$ is commutative, associative and idempotent; note that this, together with (ii), implies reflexivity and transitivity of $\Rightarrow$, whereas (ii) together with reflexivity of $\Rightarrow$ would imply idempotence of $[\!]$.

(iv) The law $x \mathbin{[\!]} y \Rightarrow x$, which would follow from (ii) and (iii).

(v) The requirement of referential transparency, which is closely related to the question whether $x \mathbin{[\!]} y - x \mathbin{[\!]} y = 0$. Note that the second approach above keeps this property.

(vi) The law $f(x \mathbin{[\!]} y) = (f\ x) \mathbin{[\!]} (f\ y)$, or the weaker $f(x \mathbin{[\!]} y) \Rightarrow (f\ x) \mathbin{[\!]} (f\ y)$. Note that the latter would follow from (i) and (ii).

(vii) The law $(x \Rightarrow y) \wedge (x \Rightarrow z)$ implies $x \Rightarrow (y \mathbin{[\!]} z)$, which would follow from (ii) and (iii).

(viii) The law $\forall\ x : f\ x \Rightarrow g\ x$ implies $f \Rightarrow g$. The other direction would follow from monotonicity of $\Rightarrow$.

We can show that these laws are mutually incompatible, even if the difficult law (v) is dropped.

Note that $[\!]/x$ stands for an "arbitrary" choice from the structure x. In particular, $[\!]/0$ describes choosing from an empty structure. What does $[\!]/0$ mean? It means that no choice is possible, i. e., it denotes the unsatisfiable specification. In particular, $[\!]/0$ satisfies $a[\!][\!]/0 = a$, meaning that having the choice between "nothing" and "something", we must choose "something".

3.6 Semantics

All expressions encountered so far are algorithms in the sense that we could build a machine to actually execute them. In many cases, however, we are interested in being able to specify our problem rather than giving a method of solving the problem, especially if we do not yet know such a method. We will allow such "unexecutable" expressions so that we are able to have the complete derivation, from the initial (formal) specification to the final algorithm in one unified framework.

In the following, we give a possible approach to the semantics of algorithmic expressions [Mee83] [B$^+$85b]. Let $\mathcal{E}$ stand for the set of algorithmic expressions. We assume that $\mathcal{E}$ is recursive, and that $\mathcal{E}$ contains a recursive subset $\mathcal{V}$ of expressions that are identified with values (e. g., "7" or "$\lambda\ x : x + 2$"). Intuitively, we can interpret an expression e in $\mathcal{E}$ as "specifying" one, or more, or possibly no, elements of $\mathcal{V}$. We define the breadth function $\mathcal{B}(e)$ to be the set $\{v \in \mathcal{V} \mid e \text{ "specifies" } v\}$. On the other hand, we can interpret e as a "task" to find some element of $\mathcal{V}$. That task might have several solutions or be impossible. Define $e \Rightarrow e'$ to mean: the task e can be solved by solving the task e'. The refinement relation $\Rightarrow$ is a subset of $\mathcal{E} \times \mathcal{E}$. We can think of $\Rightarrow$ as "may be transformed to". The refinement relation is reflexive and transitive. Interpreting an expression e as a specification of values in $\mathcal{V}$, we would expect e to specify a given $v \in \mathcal{V}$ whenever $e \Rightarrow v$. Conversely, if $v \in \mathcal{B}(e)$, then v is a solution of the task e, so we have $e \Rightarrow v$. We conclude that $\mathcal{B}(e) = \{v \in \mathcal{V} \mid e \Rightarrow v\}$.

We would like to characterize $\Rightarrow$ in terms of $\mathcal{B}$. A requirement for $e \Rightarrow e'$ is certainly $\mathcal{B}(e) \subset \mathcal{B}(e')$. But then, for any e, $e \Rightarrow [\!]/0$, which is unreasonable unless $e = [\!]/0$. This gives rise to the second requirement that $[\!]/0$ is a replacement for e only if $e = [\!]/0$. However, this requirement of "preservation of definedness" complicates the transformation rules. An alternative approach is to accept the validity of $e \Rightarrow \Box/0$, keeping in mind that $\Rightarrow$ does not exactly mean "may be replaced by". In this approach, we would check for preservation of definedness individually for transformations involving $\Rightarrow$.

Note that the meaning of $\Rightarrow$ and the derivation of further refinement rules highly depend on the chosen approach to nondeterminism.

3.7 Application to Lists

Since the (finite) *list* is a data structure with many important applications, we would like to examine some specialized list operations [Bir86]. For lists, $\hat{}$ is the function $a \mapsto [a]$, $+$ is the *concatenation* $\diamond$, and 0 is the empty list $[\,]$.

Length. The length of a list is the number of elements it contains. We denote this operation by $\sharp$.

Directed Reduction. We now introduce two new reduction operators which are closely related to $/$. $\not\rightarrow$ (left-reduce) and $\not\leftarrow$ (right-reduce) each take three arguments, and operator $\oplus$, an initial value e and a list x. They can be described by

$$(\oplus\!\not\leftarrow\! e)[a_1, a_2, \ldots, a_n] = a_1 \oplus (a_2 \oplus (\cdots (a_n \oplus e)))$$

$$(\oplus\!\not\rightarrow\! e)[a_1, a_2, \ldots, a_n] = ((e \oplus a_1) \oplus a_2) \cdots \oplus a_n.$$

Why do we need two more reduction operators? There are two main answers to this question. First, the directed reductions can be seen as implementations of the operator $/$ in which the order of evaluation is sequential. Certainly, if $\oplus$ is associative and has identity element e, then

$$\oplus/ = (\oplus\!\not\leftarrow\! e) = (\oplus\!\not\rightarrow\! e).$$

The second answer is that many more functions can be described by directed reductions than by $/$. Furthermore, although every homomorphism can be expressed as a reduction, many functions which are not homomorphisms can be described as directed reductions.

Observe that we can characterize both forms of directed reduction recursively, we have

$$\begin{aligned} (\oplus\!\not\leftarrow\! e)[\,] &= e \\ (\oplus\!\not\leftarrow\! e)([a] \diamond x) &= a \oplus (\oplus\!\not\leftarrow\! e)x \end{aligned}$$

and

$$\begin{aligned} (\oplus\!\not\rightarrow\! e)[\,] &= e \\ (\oplus\!\not\rightarrow\! e)(x \diamond [a]) &= (\oplus\!\not\leftarrow\! e)x \oplus a \end{aligned}$$

We see that $\not\leftarrow$ processes lists from right to left, and $\not\rightarrow$ from left to right. In some sense, right-reduction corresponds to *recursion*, and left-reduction corresponds to *iteration*. To make this more clear, we give an alternative recursive description of left-reduction:

$$\begin{aligned} (\oplus\!\not\rightarrow\! e)[\,] &= e \\ (\oplus\!\not\rightarrow\! e)([a] \diamond x) &= (\oplus\!\not\rightarrow\!(e \oplus a))x. \end{aligned}$$

The equivalence of the two definitions can be easily verified by induction.

Note that although $\not\leftarrow$ and $\not\rightarrow$ look similar, there may be a big difference in efficiency between them, depending on the application. Generally, when processing lists from left to right, all elements of the list have to be considered before the result can be returned. However, not all right-to-left computations must necessarily start processing at the right end of the list. Returning a result without evaluating arguments whose values are not needed is known as *lazy evaluation.* Consider for example the function $(\ll\not\leftarrow e)$, which selects the first element of a list.

$$(\ll\not\leftarrow e)[1\ldots100] = 1\ll(\ll\not\leftarrow e)[2\ldots100] = \ldots = 1,$$

so the evaluation terminates after one step, and we do not look at the rest of the list. On the other hand, $(\ll\not\rightarrow 0)$ always returns 0, but, using the second recursive definition of $\not\rightarrow$,

$$(\ll\not\rightarrow 0)[1,2,3] = \ldots = 0.$$

but in this evaluation, the complete list is traversed before the result is returned.

Formal Differentiation. To conclude this section, we would like to give an idea of how formal differentiation [Pai81,PK82,Pai86] fits into the Algorithmics framework [Mee]. In many applications we want to evaluate an expression of the form

$$f((\oplus\not\rightarrow e)[a_1, a_2, \ldots, a_n]$$

or even

$$f*[a_1, a_1 \oplus a_2, \ldots, (\oplus\not\rightarrow e)[a_1, a_2, \ldots, a_n]].$$

We would like to compute such a value incrementally, i. e., (for the first expression) in the form of

$$(\otimes\not\rightarrow(f\ e))[a_1, a_2, \ldots, a_n].$$

We can do so by finding an operator $\otimes$ such that

$$f(x \oplus a) = (f\ x) \otimes a,$$

since $f\circ(\oplus\not\rightarrow e)$ satisfies the recursive equations

$$\begin{aligned} (f\circ(\oplus\not\rightarrow e))[\,] &= f\ e \\ (f\circ(\oplus\not\rightarrow e))(x \diamond a) &= ((f\circ(\oplus\not\rightarrow e))x) \otimes a, \end{aligned}$$

which are also solved by $\otimes\not\rightarrow(f\ e)$. Although such an operator $\otimes$ cannot necessarily be found for all f and $\oplus$, it is always possible to find one for (f, id), defined by

$$y \otimes z = (f(\pi_2 y \oplus z), \pi_2 y \oplus z).$$

If (f,id) can be computed efficiently, then so can $f = \pi_1(f,\mathrm{id})$, where π_i returns the i-th component of a tuple.

4 The RAPTS Transformational Programming System

RAPTS [Pai86] is a system for the fully mechanical transformation of problem specifications to efficient RAM code. RAPTS uses stepwise refinement by successive application of correctness-preserving transformations to automate program design, verification, and analysis.

The input language for the RAPTS compiler is SQ+, an abstract functional specification language based on finite set theory, which can be shown to express any partial recursive function in a *fixed-point normal form.* SQ+ is essentially a functional subset of SETL [SDDS86] augmented with fixed-point operations. The compiler produces sequential RAM code in the C language.

RAPTS uses several techniques to produce efficient target code which correspond to different phases of the compiler. The first phase of the compiler translates expressions from fixed-point normal form into an *iterative fixed-point form.* The next phase applies *finite differencing* to the code. The final phase performs *data structure selection* for a RAM. The compiler is biased towards greedy strategies.

In addition to the compiler, RAPTS provides several other functions such as maintaining transformation libraries, parsing/unparsing etc. A prototype of the system was implemented in SETL.

The RAPTS methodology aims to be practical, emphasizes automation, and focuses on a subclass of determinate, tractable problems that compute finite sets. This subclass captures a wide variety of problems arising in practice.

We will discuss the language SQ+ and the techniques used in the three phases of compilation.

4.1 The Specification Language SQ+

SQ+ [PH87,CP88] is a very-high-level functional problem specification language. It is a functional subset of SETL, consisting of expressions over boolean and integer data types and finite-set expressions, enhanced with fixed-point expressions. SQ+ provides function abstraction. The semantics of SQ+ is defined operationally in terms of a lower-level imperative set-theoretic machine language (see tables).

Most SQ+ expressions conform to well-known mathematical notations, with the exception of maps. We regard a map as a finite set of ordered pairs that maps a domain set to a range set. Thus, a map can be single-valued or multi-valued. Function retrieval is denoted by $g(x)$, while multi-valued map retrieval is denoted by $g\{x\}$.

SQ+ expressions and lower-level constructs are described in the tables below, and their *set-theoretic complexity* is given. The complexity measure can be based on efficient hash-table implementations of sets and maps. We assume that a single hash operation on a data item with unit-space storage takes unit time, and that searching through a set takes time proportional to the cardinality of the set. In the tables, let Q and T be any stored sets, and let g be a map.

The sublanguage SQ (SQ+ without fixed-point expressions) was shown to have at least the expressive power of Relational Algebra. To express transitive closure, we need to add the *fixed-point expressions* $\mathbf{LFP}_{\subseteq,S}(F)$ and $\mathbf{GFP}_{\subseteq,B}(F)$ (for least and greatest fixed point, respectively) to the expressions in the tables.

In addition, we allow specifications either given in fixed-point normal form, i. e.,

$$\mathbf{the}\ Q : S \subseteq Q \mid Q = F(Q)\ \mathbf{minimizing}\ Q,$$
$$\mathbf{the}\ Q : Q \subseteq B \mid Q = F(Q)\ \mathbf{maximizing}\ Q,$$

or in the more general form

$$\mathbf{the}\ Q : S \subseteq Q \mid K(Q)\ \mathbf{minimizing}\ Q,$$
$$\mathbf{the}\ Q : Q \subseteq B \mid K(Q)\ \mathbf{maximizing}\ Q.$$

Expression/Operation	Definition	Complexity
$Q := \{\}$	assign empty set	O(1)
Q **with** $:= x$	set element addition	O(1)
Q **less** $:= x$	set element deletion	O(1)
$x \in Q$	set membership test	O(1)
$\ni Q$	arbitrary choice	O(1)
$\exists\ x \in Q$	test for empty set with arbitrary assignment to x	O(1)
$g\{x\}$	image set of x under g	O(1)
$g(x)$	y, if $g\{x\} = \{y\}$ Ω (undefined), otherwise	O(1)
$g\{x\} := \{\}$	make image set empty	O(1)
$g\{x\}$ **with** $:= y$	add element to image set	O(1)
$g\{x\}$ **less** $:= y$	delete element from image set	O(1)
$y \in g\{x\}$	image set membership test	O(1)
$g(x) := \Omega$	remove x from **domain** g	O(1)
$g(x) := z$	make $g(x) = z$	O(1)
domain g	elements with nonempty g-image	O(1)

(**for** $x \in Q$)
Block(x)
end for
	execute Block for each element x in a copy of Q	O($\sharp Q *$ cost(Block(x)))

Table 1: Elementary SQ operations

Expression/Operation	Definition	Complexity
range g	set of all images under g	O($\sharp g$)
$\sharp Q$	set cardinality	O($\sharp Q$)
$g[Q]$	image of Q under g	O($\sharp Q$)
$\{x \in Q \mid K(x)\}$	set former	O($\sharp Q *$ cost($K(x)$))
$\{\mathrm{e}(x) : x \in Q\}$	set former	O($\sharp Q *$ cost(e(x)))
$Q \cap T$	set intersection	O($\sharp Q$)
$Q - T$	set difference	O($\sharp Q$)
$Q \cup T$	set union	O($\sharp Q + \sharp T$)
$\exists\ x \in Q \mid K(x)$	existential quantifier	O($\sharp Q *$ cost($K(x)$))
$\forall\ x \in Q \mid K(x)$	universal quantifier	O($\sharp Q *$ cost($K(x)$))
$Q \times T$	cartesian product	O($\sharp Q * \sharp T$)
min$/Q$	minimum value in Q	O($\sharp Q$)
$Q := T$	copy set T to set Q	O($\sharp Q$)

Table 2: Nonelementary SQ operations

These specifications are an extension of SQ+; RAPTS tries to transform them to fixed-point expressions.

Although SQ+ is Turing-complete, it is practical to focus mainly on tractable problems expressed by a highly restricted subset of SQ+.

4.2 Fixed-Point Transformations

Given a specification of the form

$$\textbf{the } Q : S \subseteq Q \mid K(Q) \textbf{ minimizing } Q,$$

RAPTS tries to rewrite the expression $K(Q)$ in an equational form

$$Q = F(Q).$$

We are now looking for a fixed point of F. With the specification in this form, it is easier to check the conditions of the fixed-point transformations mechanically. To rewrite $K(Q)$ in an equational form, RAPTS uses a uniformly terminating *rewrite system*, whose current implementation is ad hoc but captures a number of frequently occurring problems. If the conditions of the fixed-point transformations cannot be verified, the compilation terminates.

Whenever various conditions on h (monotone, inflationary at S, i. e., $h(S) \supseteq S$), are satisfied, the specification

$$\textbf{the } P : P \subseteq S \mid h(P) = P \textbf{ minimizing } P$$

is well-defined and can be implemented using the following imperative code,

```
$P := S$
(**converge**)
        $P := h(P)$
**end**;
```

which converges after a finite number of steps. The analog holds for the dual specification

$$\textbf{the } P : P \subseteq B \mid h(P) = P \textbf{ maximizing } P.$$

Such an implementation may be inefficient for two reasons. First, the new approximation of P is completely recomputed and copied at each iteration, although it might differ only slightly from its previous value. Second, depending on the particular h, the iterative step may be biased towards a particular search strategy; this also makes the analysis difficult. We would like to iterate nondeterministically in a way that allows us to take advantage of only slight changes of P.

Cai and Paige [CP88] developed a theory for fixed-point computations which is very general and applies to partially ordered sets and semilattices. We restrict ourselves to presenting applications to collections of sets, which are lattices. This restriction makes sense since the set is one of the simplest and most widely used data types, and the basic set operations frequently satisfy the conditions of the transformations.

In their paper [CP88], they develop an algebraic approach to nondeterministic iteration in fixed-point computation. A partial function Δ is called a *workset function* if $\Delta(Q, P)$ is either undefined or empty if and only if $Q \subseteq P$. A partially defined function δ is called an *increment function* if $\delta(P, z)$ is either undefined or $\delta(P, z) \supseteq P$. The two functions are said to be *feasible* relative to h if an increment of S within the workset strictly increases S but not beyond $S \cup f(S)$. Within this framework, the following transformation can be proved correct:

```
$P :=$ $\mathbf{LFP}_{\subseteq,W}(f)$
      $\updownarrow$
$P := W$;
(while $\exists\ z \in \Delta(f(P), P)$)
    $P := \delta(P, z)$;
end
```

Whether the generated code is efficient depends on the choice of Δ and δ. It is easy to find such functions for a powerset T and a monotone function f by choosing

$$\begin{aligned} \Delta(Q,P) &= \begin{cases} \{\} & \text{if } Q \subseteq P \\ \{Q\} & \text{otherwise, and} \end{cases} \\ \delta(P,z) &= P \cup z. \end{aligned}$$

The transformation leads to the usual iteration

```
$P := W$;
(while $f(P) \supset P$)
    $P := f(P)$;
end
```

If we choose $\Delta(Q,P) = Q - P$ and $\delta(P,z) = P$ **with** z, we get the following nondeterministic iteration:

```
$P := W$;
(while $\exists\ z \in f(P) - P$)
    $P$ with $:= z$;
end
```

It is possible to refine the basic transformation to compute fixed points for various functions f. Under certain conditions, for example, $f(S)$ and $S \cup f(S)$ have the same fixed point, so that

```
$P :=$ $\mathbf{LFP}_{\subseteq,W}(S \cup f(S), S)$
      $\updownarrow$
$P := W$;
(while $\exists\ z \in \Delta(f(P), P)$)
    $P := \delta(P, z)$;
end
```

The transformation can also be refined with respect to different underlying data types such as decomposable lattices. The theory can be generalized to deal with systems of fixed-point equations.

The code generated by our transformations might still be inefficient, since the computation of $\Delta(f(P), P)$ could be expensive and is performed with each iteration. The same might hold for the assignment $P := \delta(P, z)$. We would like to apply *finite differencing* to implement fixed-point computations more efficiently.

4.3 Finite Differencing

Finite differencing [Pai81,PK82,Pai86] is a technique used to avoid recomputations of expressions of the form

$$f(x_1, \ldots, x_n)$$

and replace them with less expensive incremental computations.

We can avoid such recomputations by maintaining the invariant

$$E = f(x_1, \ldots, x_n)$$

at the program point where the f is evaluated; this allows us to replace the computation of f with its stored value E. That is, we

1. *establish* the invariant by evaluating f into E on entry to the program region where the value of f is needed;

2. update E within that program region whenever any of the parameters $x_1, \ldots, x_n$ of f are modified; this update code is called *difference code* for E with respect to the modifications $dx_1, \ldots, dx_n$;

3. replace within the program region all occurrences of $f(x_1, \ldots, x_n)$ by E.

This technique is called *finite differencing*, a generalization of strength reduction. Of course, finite differencing is worthwhile only if the cumulative cost of executing the difference code for expression E within the program after finite differencing is applied is less than the cost of repeatedly evaluating f in the original program. The idea is to recognize those expressions that can be maintained inexpensively as invariants. RAPTS stores a finite collection of *elementary differentiable expressions* and their associated blocks of difference code guaranteed to be inexpensive.

Whenever an expression can be recognized as being composed of elementary differentiable expressions, then finite differencing can be applied. In order to expose hidden elementary differentiable expressions and to make the code more regular, expressions are placed into a normal form using another rewrite system. This rewrite system performs a variety of minor symbolic manipulations such as turning set difference and intersection into equivalent set formers.

The difference code for a modification dx generally consists of a piece of code before the modification, called *predifference code*, and a piece after the modification, called *postdifference code*. We write $\partial^- < dx >$ and $\partial^+ < dx >$, respectively. The predifference and postdifference code blocks can modify only the variable E and local variables. Note that the difference code is not unique.

It is interesting to consider collections of equalities $E_1 = f_1, \ldots, E_k = f_k$, in which each expression f_j depends only on variables $x_1, \ldots, x_n$ and $E_1, \ldots, E_{j-1}$. For the purpose of efficiency, we would like to maintain and exploit all of these equalities as invariants within a program region B. The *differential* of $E_1, \ldots, E_k$ with respect to B is denoted by $\partial\{E_1, \ldots, E_k\} < B >$. It is obtained from B by recursively applying the following transformations:

1. Replace each modification dx occurring in B with

$$\partial\{E_2, \ldots, E_k\} < \partial^- E_1 < dx > \;\; dx \;\, \partial^+ E_1 < dx >>,$$

where no new occurrences of f_1 are introduced within the difference code for the remaining invariants $E_2, \ldots, E_k$, and all occurrences of f_1 can be replaced by E_1. We call E_1 the *minimal invariant* for the differential $\partial\{E_1, \ldots, E_k\} < dx >$.

2. Replace all occurrences of f_j by the corresponding E_j within the rest of B

This gives us the general *chain rules* for collective predifference and postdifference code blocks, where E_1 must be minimal invariant:

$$\begin{aligned} \partial^-\{E_1,\ldots,E_k\} < dx > &= \partial\{E_2,\ldots,E_k\} < \partial^- E_1 < dx >> \\ & \quad \partial^-\{E_2,\ldots,E_k\} < dx > \\ &\text{and} \\ \partial^+\{E_1,\ldots,E_k\} < dx > &= \partial^+\{E_2,\ldots,E_k\} < dx > \\ & \quad \partial\{E_2,\ldots,E_k\} < \partial^+ E_1 < dx >> . \end{aligned}$$

Minimal invariants can be found by examining the *data dependency dag* for $f_1,\ldots,f_k$.

In order to determine whether finite differencing actually improves the performance of the code, the set-theoretic cost of the resulting program is determined *before* applying any transformations. This *syntactic* analysis involves three components: the cost of establishing the invariants before the code block, the cost of maintaining them within the block, and the remaining costs. The concept of *continuity* is introduced to argue about the costs of maintaining invariants. *Strong continuity* means that the worst case cost of reestablishing an invariant after a single parameter is modified is bounded by a constant. *Weak continuity* means that the cumulative cost of maintaining an invariant relative to all modifications to the parameter is bounded by the cost of establishing the invariant plus the number of modifications to the parameter. Certain continuity properties hold for the composition of continuous functions. Although the concept of continuity helps only in the case of linear complexity, it can be generalized to arbitrary polynomial complexities.

Some additional techniques can be used to speed up the code obtained after finite differencing by a constant factor:

- A collection of invariants can easily be established in a naive way by separately initializing each E_i. RAPTS uses a *stream-processing* technique to efficiently establish several mutually dependent invariants in few passes. This gives a constant-factor speedup.

- Certain collections of invariants can be established in fewer loops using two other loop-combining techniques called *vertical* and *horizontal fusion.*

- Finite differencing, streaming and fusion increase data independence and possibly introduce useless code. *Useless-code elimination* is used to achieve another constant-factor speedup at the end of the finite-differencing phase.

4.4 Data-Structure Selection

The code obtained after the first and second phases of compilation still uses sets and maps as basic data types; its performance is analyzed in terms of the set-theoretic complexity measure. The third phase of the compiler, whose implementation is not yet complete, implements these sets and maps using conventional storage structures on a *uniform-cost sequential RAM* [PH87]. The RAM code generated is guaranteed to execute with the same worst-case asymptotic space and time RAM complexities as the set-theoretic complexities as the set-machine code.

This last compilation phase consists of the following two steps:

1. All non-elementary set-theoretic operations in the program are implemented in terms of the elementary operations shown in the table. The resulting program is said to be in *set-machine-code normal form.* This step is straightforward and does not change the complexity of the code.

2. We try to implement each elementary set operation in terms of conventional RAM operations such that the asymptotic worst-case time and space complexities on the RAM are as good as the set-theoretic complexities.

These performance objectives cannot always be achieved. In that case, the compiler would apply heuristics such as representing sets as hash tables or search trees.

RAPTS tries to avoid costly copy operations and hidden costs of garbage collection by imposing various *copy avoidance* and *deallocation conditions.*

Depending on the kind of operations required for each *stored set* (maps are stored as sets) in the program, RAPTS chooses an appropriate internal representation for that set. Possible representations are:

- *unbased* set, a doubly-linked list with pointers to the first and last list cell;
- *local* set, consisting of a table of all possible values of the set, called *base*, pointers linking the table elements currently in the set, and pointers to the first and last element;
- *sparse* set, a doubly-linked list where each cell points to the corresponding element in the base table of a local set.

The RAPTS data-structure selection phase can be viewed as a highly constrained variant of the SETL data-structure selection component.

5 An Comparative Evaluation

The three approaches to transformational programming we have presented cannot be compared easily. In this section, we first present a framework of common criteria by which transformation systems can be evaluated. Then we compare the three systems with respect to these criteria.

5.1 Criteria for Evaluating Transformation Systems

We group our criteria in general criteria, system aspects, and language aspects.

5.1.1 General Criteria

objective: The primariy goal of most transformation systems is the *general support for program modification.* This includes the *optimization* of control structures, the *implementation* of data structures, the adaption of given programs to particular styles of programming, and the *generation* of new transformation rules. Further goals are *program synthesis,* the derivation of a program from a specification, *program adaption* to particular environments or languages, *program description,* and (deduction-oriented) *verification.*

problem domain: An important criterion is the problem domain a transformation system can handle. Some systems restrict their problem domain to provide a higher degree of automation. Others handle a wide range of problems but require that transformations be selected by the user.

extensibility: Several parts of a transformation systems may be *extensible;* the library of transformation rules may be extended, or a new specification or programming language may be incorporated.

5.1.2 System Aspects

organization and types of transformations: Most systems provide a *predefined collection of transformation rules,* which may later be extended. There are two contrary methods for maintaining the library of transformations of the system: the *catalog approach* and the *generative set approach.* A catalog is a large collection of domain-specific knowledge. On the other hand, a generative set is a small set of powerful elementary rules, from which additional rules can be derived. Recently, there seems to have been a tendency towards the latter.

form of transformation rules: In general, transformation rules consist of input template, output template, and applicability condition. The input and output templates may be related by *equivalence* or *descendance.* Rules may be *procedural* (algorithmic) or *schematic.* Typically, procedural rules are used as *global* rules, whereas schematic rules are *locally applicable refinement rules.*

transformation process and mechanization: Systems may be fully manual, semi-automatic, or fully automatic. A manual system is practical only in connection with powerful transformation rules. Semi-automatic systems may require user assistance on certain, hard transformations. Fully automatic systems often restrict the problem domain and apply domain-specific heuristics.

system support: Besides the system component that deals with transformations, a system may provide facilities for prettyprinting (often in connection with parsing and unparting) and for documentation of the development process (ranging from low-level bookkeeping to browsing the decision tree representing the development histor).

5.1.3 Language Aspects

specification and target languages: The choice of languages used is closely related to the problem domain of a transformation system. A system may provide separate *specification* and *target* languages, or allow the development of programs within a single, wide-spectrum language. In the former case, the specification language is often a very-high-level descriptive language, and the target language a conventional programming language, whereas a wide-spectrum language accomodates all levels of programming from (non-executable) specifications down to programs. Systems may use a third language for formulating *transformation rules* or *transformation algorithms.*

data types: Again, the range of data types offered by a system is linked to its problem domain. Transformation systems may have a fixed set of data types, e. g., sets and maps, or support user-defined algebraic data types.

nondeterminism: Nondeterminism is an important aspect of a transformation system. It allows the programmer to avoid over-specifying a problem; this may keep paths to an efficient implementation open. Nondeterminism may be explicit in form of a *choice operator,* or implicit in form of *arbitrary selection,* e. g., from a set.

mathematical soundness: Depending on its problem domain, a transformation system may depend on certain mathematical methods. It is desirable for these methods to be supported by a sound theoretical basis.

5.2 Objective and Problem Domain

The goal of the CIP system is to deal with general programming problems and to support the user as much as possible in mechanizable tasks. There is no intent to mechanize any decisions during the transformation process. CIP is so general that its object language, CIP-L, does not contain any predefined data types. All data types are user-defined; some type definitions might be found in a standard library.

The Algorithmics approach tries to put programming into a mathematical framework based on function application. While a mathematician manipulates formulas, a programmer manipulates programs in the same way, either using known transformations or proving new results. It is a pure pencil-and-paper approach, no automation takes place. This sets Algorithmics apart from the two other approaches, which strongly emphasize system support.

RAPTS is more pragmatically oriented. Its problem domain is restricted to fixed-point expressions over powerset lattices, which it automatically compiles to efficient, often linear-time implementations. RAPTS also focuses on easy runtime analysis of the code it produces.

5.3 Extensibility

CIP-S is extensible in several respects. The object language is not fixed; it can be replaced, or it can be extended by introducing new constructs by transformation to given ones. The library of transformation rules can be extended.

Since Algorithmics is a mathematical framework, it can be extended arbitrarily. New types, operations, and transformations can be added.

As a generalization of sets and maps, Cai and Paige consider fixed-point transformations for abstract functions defined on lattice-theoretic data types. Finite differencing is defined abstractly for any data type. The rewrite systems in RAPTS can be extended to cover additional problem specifications and differentiable expressions. New transformations can be added as well.

5.4 Transformation Libraries and Rules

In general, transformation rules consist of input template, output template, and applicability condition.

In CIP, input and output templates are program schemes and may include context parameters. The rules are organized as a generative set, i. e., a small set of powerful rules which may be used in transformation expressions. The set of rules is extensible, problem-specific and frequently used rules may be added.

Transformations in Algorithmics are rules for the manipulation of applicative expressions. The rules have the flavor of mathematical identities. Algorithmics tries to keep rules as general as possible and avoid applicability conditions.

RAPTS uses different kinds of transformations at the various phases of compilation. Fixed-point expressions and set expressions are transformed to appropriate normal forms using rewrite systems. Other transformations lead from applicative to iterative form, and to differential form. The rules are implemented in large catalogs.

The form of transformation rules has to do with the extent of mechanization a system provides. Since Algorithmics is non-automatic, its rules are kept as general as possible, thus making it easier for the user to manipulate expressions and formulas. More specific rules with complicated applicability conditions as in CIP-L or RAPTS call for extensive system support in finding appropriate rules and checking conditions.

5.5 Mechanization and System Support

From a software engineering point of view, it is important that a system automate as many functions as possible and leave to the programmer only tasks which require intuition.

CIP tries to provide exactly this. CIP-S keeps track of the complete development history of a program. It assists the user in verifying applicability conditions of transformations and rewrites program pieces after applying transformations.

Algorithmics, on the other hand, provides no mechanization. Using the Algorithmics methodology can be compared to a mathematician's work when simplifying arithmetic formulas.

By dealing with more specific problems, RAPTS is able to compile fully automatically a specification to an efficient implementation. In each compilation phase, program pieces are rewritten mechanically.

There is a tradeoff between generality and mechanization. By restricting the problem domain, specific knowledge can be incorporated into the system and used to automate design decisions.

5.6 Specification and Target Languages

CIP-L, the standard object language of CIP, is a wide-spectrum language providing constructs from specification level down to control-oriented level. Since CIP-L is a scheme language whose data types can be adapted to the problem being solved, it is fully general. One should note that the object language in CIP is not fixed; any algebraically defined language may be used.

The Algorithmics language consists of applicative expressions over data objects and functions. New objects and operations may be defined in terms of given ones using equalities. Functions may be polymorphic.

RAPTS distinguishes between specification language and target language. Problems are specified in SQ+, a SETL-like functional language with fixed-point expressions. The RAPTS compiler produces RAM code. Intermediate steps use SQ+ in which descriptive constructs have been replaced by iterative ones.

In a system such as RAPTS, which fully mechanically compiles a specification to a program, it is a good idea to have separate specification and target languages. In CIP and Algorithmics design decisions are made by the user. Gradually, parts of the program are replaced by lower-level constructs as we approach an implementation, while other parts remain unchanged. A single object language allows different stages of the development to coexist in one program.

5.7 Data Types

CIP allows the user to specify any algebraic type by stating its objects and operations and the properties that hold between them. Such a specification is effective only if the user, in addition, implements a model of the type, i. e., explicitly programs the operations such that the specified properties hold. Types may be combined hierarchically.

A similar concept is used by the Algorithmics group. Types are defined by specifying its objects, operations, and laws. Algorithmics does not require types to be implemented, it requires the existence of a term-generated model of a type. A type hierarchy can be formed adding more laws to a given type.

RAPTS uses sets and maps as data types. Many practical problems can be formulated in terms of sets, maps, and fixed-point expressions. This restriction makes it possible to provide automatic implementation of the data types.

5.8 Nondeterminism

CIP-L provides an arbitrary-finite-choice operator which applies to (non-functional) values. To avoid semantic difficulties, finite choice between functions is not permitted. The additional comprehensive-choice expression nondeterministically chooses some object that satisfies a predicate.

The Algorithmics group has not yet decided whether to allow an explicit choice operator or to allow nondeterminism indirectly through set construction only. It is likely that a choice operator will be incorporated in the language. This choice operator could be an arbitrary-choice operator as in CIP-L or an operator that denotes some definite, but unspecified, selection operator.

In RAPTS, nondeterminism is expressed through arbitrary selection from a set and arbitrary search through a set. This avoids restricting the order in which the solution set is generated.

The advantage of avoiding explicit nondeterminism is a simpler semantics of the object language. However, function and object definitions need to be extended beyond equational predicates to general set membership predicates. Letting the choice operator denote a definite, unspecified selection operator also results in simpler semantics, but certain laws (e. g., commutativity) may no longer be valid. We believe that the third approach, i. e., allowing an arbitrary-choice operator, is more appropriate, because one can develop a system of desirable properties of the choice operator. Semantic problems can be avoided by restricting the choice operator to a choice between non-functional values.

5.9 Mathematical Soundness

The CIP group made a big effort to put their work on a solid mathematical basis. Part of the activities in the CIP project was the development of transformational semantics for CIP-L and a logical calculus of program transformations for CIP-S. In addition, the group formally developed the existing prototype of CIP-S using the CIP methodology.

Mathematical soundness is an important issue in the work of the Algorithmics group. Although certain decisions about nondeterminism and semantics have not yet been made, they are considered to be very important.

The theories underlying RAPTS, i. e., finite differencing and fixed-point theory for lattices, have been developed formally. As far as the system itself goes, the emphasis is more on pragmatic issues than on formality.

5.10 Conclusion

It is difficult to compare such different approaches to transformational programming. An evaluation would highly depend on what we would establish as the goal of the transformational methodology. If we want to develop a mathematical discipline of programming independent of pragmatic issues such as large scale programming, the direction chosen by the Algorithmics group seems appropriate. If, on the other hand, we know that we are dealing mainly with a restricted class of problems that we want to treat in a pragmatic, mechanized way, the RAPTS approach is well-suited. Considering again our initial motivation, providing a transformational component of a general prototyping system, we would look for a system which both has a general problem domain and offers extensive system support as far as the user environment is concerned. We feel that the CIP group is facing the right direction as far as the desgin of the wide-spectrum language and the transformation system goes. However, pragmatic issues should be given more weight; the system should provide a large library of frequently used standard data types, a comfortable (implemented) user environment,

criterion	**CIP**	**Algorithmics**	**RAPTS**
objectives	general programming	mathematical framework	supercompiler
domain	general	functional	fixpoint expressions
extensibility	languages, rules	unrestricted	rules
transformations	generative set	manual	catalog
rules	input, output, condition	general	rewrite rules
mechanization	semi-automatic	none	automatic
system support	limited	none	extensive
languages	wide-spectrum, schemes	functional	SETL with fixpoints
types	abstract, algebraic	algebraic	sets and maps
nondeterminism	arbitrary choice	various	selection from set

Table 3: Evaluation of Transformation Systems (Overview)

and a library of automated transformations such as in RAPTS for certain well-understood problem domains.

Acknowledgments

I would like to thank Robert Dewar for initial suggestions and guidance throughout the survey; Bob Paige for valuable comments on the paper; and my friends and fellow students Franco Gasperoni and Marco Pellegrini for their moral support.

References

[B⁺85a] F. L. Bauer et al. *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1985.

[B⁺85b] Richard Bird et al. A common basis for algorithmic specification and development. IFIP WG2.1 Working paper ARK-3, 1985.

[B⁺87a] F. L. Bauer et al. *The Munich Project CIP. Volume II: The Program Transformation System CIP-S*, volume 293 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1987.

[B⁺87b] Richard S. Bird et al. Two exercises found in a book on algorithmics. In Lambert Meertens, editor, *Program Specification and Transformation*. Elsevier Science Publishers B.V. (North-Holland), 1987.

[Bir86] Richard S. Bird. An introduction to the theory of lists. Technical report, Oxford University Computing Laboratory. Programming Research Group, 1986.

[CP88] Jiazhen Cai and Robert Paige. Program Derivation by Fixed-Point Computation. Technical report, IBM Research Division, T. J. Watson Research Center, 1988. To appear in revised form in *Science of Programming.*

[D⁺90] R. Dewar et al. Reference manual for the Griffin prototyping language. In progress, December 1990.

[Gab88] Gabriel, R. (editor). Draft report on requirements for a common prototyping system, November 1988. Common Prototyping Working Group.

[Gor79] Michael J. C. Gordon. *The Denotational Description of Programming Languages. An Introduction.* Springer-Verlag, New York, 1979.

[Mee] Lambert Meertens. Formal differentiation. A page from a book on algorithmics.

[Mee83] Lambert Meertens. Algorithmics. Towards programming as a mathematical activity. In *Mathematics and Computer Science. Proc. CWI Symp.*, November 1983.

[Mee84] Lambert Meertens. Some more examples of algorithmic developments. IFIP WG2.1 Working paper ADP-7, 1984.

[Pai81] Robert Paige. *Formal Differentiation. A Program Synthesis Technique.* UMI Research Press, Ann Arbor, Michigan, 1981.

[Pai86] Robert Paige. Programming with invariants. *IEEE Software*, 1986.

[PH87] Robert Paige and Fritz Henglein. Mechanical translation of set theoretic problem specifications into efficient RAM code — A case study. *Journal of Symbolic Computation*, (4):207–232, 1987.

[PK82] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.

[PS83] H. Partsch and R. Steinbrüggen. Program transformation systems. *Computing Surveys*, 15(3):199–236, September 1983.

[SDDS86] J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL.* Springer-Verlag, 1986.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Massachusetts, 1977.

[WJS87] R. J. Watro, D. M. Johnson, and R. M. Smaby. Transformational program development: An examination of the Munich CIP approach. The MITRE Corporation, March 1987.